2

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Ada Compiler Validation Summary Report: SYSTEAM KG
SYSTEAM Ada Compiler SUN/SUNOS 1.81, Sun 3/60 (host & target), 891102I1.10200

5. TYPE OF REPORT & PERIOD COVERED
2 Nov 89 - 1 Dec 90

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
~~IABG~~,
Ottobrunn, Federal Republic of Germany.

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS
IABG,
Ottobrunn, Federal Republic of Germany.

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office)
IABG,
Ottobrunn, Federal Republic of Germany.

15. SECURITY CLASS (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

SYSTEAM KG, SYSTEAM Ada Compiler SUN/SUNOS Version 1.81, IABG, West Germany, Sun 3/60 under SunOS, Version 4.0.3 (host & target), ACVC 1.10

AVF Control Number: IABG-VSR-048

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #891102I1.10200
SYSTEAM KG
SYSTEAM Ada Compiler SUN/SUNOS 1.81
Sun 3/60 Host and Target

Completion of On-Site Testing:
2nd November 1989

Prepared By:
IABG mbH, Abt SZT
Einsteinstr 20
D8012 Ottobrunn
West Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: SYSTEAM Ada Compiler SUN/SUNOS
Version 1.81

Certificate Number: #891102I1.10200

Host: Sun 3/60 under SunOS, Version 4.0.3

Target: Sun 3/60 under SunOS, Version 4.0.3

Testing Completed Thursday, 2nd November 1989 Using ACVC 1.10

This report has been reviewed and is approved.

IABG mbH, Abt SZT
Dr S. Heilbrunner
Einsteinstr 20
D8012 Ottobrunn
West Germany

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

John P. Solomond

Ada Joint Program Office
Dr John Solomond
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report ~~(VSR)~~ describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability ~~(ACVC)~~. An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

(KR)

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by IABG mbH, Abt SZT according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed Thursday, 2nd November 1989 at SYSTEAM KG, Karlsruhe.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from

IABG mbH, Abt. SZT
Einsteinstr 20
D8012 Ottobrunn

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.
4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.

Ada Commentary An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.

Ada Standard ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant The agency requesting validation.

AVF The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.

AVO The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

Compiler A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

Failed test An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host The computer on which the compiler resides.

Inapplicable test
An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test
An ACVC test for which a compiler generates the expected result.

Target
The computer which executes the code generated by the compiler.

Test
A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn test
An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: SYSTEAM Ada Compiler SUN/SUNOS Version 1.81

ACVC Version: 1.10

Certificate Number: #891102I1.10200

Host And Target Computer:

Machine :	Sun 3/60
Operating System :	SunOS, Version 4.0.3
Memory Size :	8 MB

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

2) The compiler correctly processes tests containing loop

statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

1) This implementation supports the additional predefined types SHORT_INTEGER, SHORT_FLOAT, LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

4) No exception is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

5) No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

1) The method used for rounding to integer is round to even. (See tests C46012A..Z (26 tests).)

2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z (26 tests).)

3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An imp⌞ementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

This implementation evaluates the 'LENGTH of each constrained array subtype during elaboration of the type declaration. This causes the declaration of a constrained array subtype with more than INTEGER'LAST (which is equal to SYSTEM.MAX_INT for this implementation) components to raise CONSTRAINT_ERROR. However the optimisation mechanism of this implementation suppresses the evaluation of 'LENGTH if no object of the array type is declared depending on whether the bounds of the array are static, the visibility of the array type, and the presence of local subprograms. These general remarks apply to points (1) to (5), and (8).

1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception if the bounds of the array are static. (See test C36003A.)

2) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components if the bounds of the array are not static and if the subprogram declaring the array type contains no local subprograms. (See test C36202A.)

3) CONSTRAINT_ERROR is raised when an array type with INTEGER'LAST + 2 components is declared if the bounds of the

array are not static and if the subprogram declaring the array type contains a local subprogram. (See test C36202B.)

4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises CONSTRAINT_ERROR when the array type is declared if the bounds of the array are not static and if there are objects of the array type. (See test C52103X.)

5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the array type is declared if the bounds of the array are not static and if there are objects of the array type. (See test C52104Y.)

6) In assigning one-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

7) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

8) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises CONSTRAINT_ERROR when the array type is declared if the bounds of the array are not static and if there are objects of the array type. (See test E52103Y.)

f. Discriminated types.

1) In assigning record types with discriminants, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

1) In the evaluation of a multi-dimensional aggregate, the test results indicate that all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

2) In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. However this implementation raises USE_ERROR upon creation of a file for unconstrained array types.(See tests AE2101H, EE2401D, and EE2401G.)

3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)

5) Modes IN_FILE, OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K AND CE2102Y.)

8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

9) Overwriting to a sequential file truncates the file to the last element written. (See test CE2208B.)

10) Temporary sequential files are given names and deleted when closed. (See test CE2108A.)

11) Temporary direct files are given names and deleted when closed. (See test CE2108C.)

12) Temporary text files are not given names. (See test CE3112A.)

13) More than one internal file can be associated with each external permanent (not temporary) file for sequential files when reading only or writing only. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

14) More than one internal file can be associated with each external permanent (not temporary) file for direct files when reading only or writing only. (See tests CE2107F..H (3 tests), CE2110D AND CE2111H.)

15) More than one internal file can be associated with each external permanent (not temporary) file for text files when reading only or writing only. (See tests CE3111A..B (2 tests), CE3111D..E (2 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 264 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 159 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 14 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT			TEST CLASS				TOTAL
	A	B	C	D	E	L	
Passed	129	1132	2058	17	27	46	3409
Inapplicable	0	6	257	0	1	0	264
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	201	591	566	245	172	99	161	331	137	36	252	325	293	3409
N/A	11	58	114	3	0	0	5	1	0	0	0	44	28	264
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	68	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

```
E28005C   A39005G   B97102E   C97116A   BC3009B   CD2A62D
CD2A63A   CD2A63B   CD2A63C   CD2A63D   CD2A66A   CD2A66B
CD2A66C   CD2A66D   CD2A73A   CD2A73B   CD2A73C   CD2A73D
CD2A76A   CD2A76B   CD2A76C   CD2A76D   CD2A81G   CD2A83G
CD2A84N   CD2A84M   CD50110   CD2B15C   CD7205C   CD2D11B
CD5007B   ED7004B   ED7005C   ED7005D   ED7006C   ED7006D
CD7105A   CD7203B   CD7204B   CD7205D   CE2107I   CE3111C
CE3301A   CE3411B
```

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 264 tests were inapplicable for the reasons indicated:

a. The following 159 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C241130..Y (11 tests)   C357050..Y (11 tests)
C357060..Y (11 tests)   C357070..Y (11 tests)
C357080..Y (11 tests)   C358020..Z (12 tests)
C452410..Y (11 tests)   C453210..Y (11 tests)
C454210..Y (11 tests)   C455210..Z (12 tests)
C455240..Z (12 tests)   C456210..Z (12 tests)
C456410..Y (11 tests)   C460120..Z (12 tests)

b. C34007P and C34007S are expected to raise CONSTRAINT_ERROR. This implementation optimizes the code at compile time on lines 205 and 221 respectively, thus avoiding the operation which would raise CONSTRAINT_ERROR and so no exception is raised.

c. C41401A is expected to raise CONSTRAINT_ERROR for the evaluation of certain attributes, however this implementation derives the values from the subtypes of the prefix at compile time as allowed by 11.6 (7) LRM. Therefore elaboration of the prefix is not involved and CONSTRAINT_ERROR is not raised.

d. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C   C45304C   C45502C   C45503C   C45504C
C45504F   C45611C   C45613C   C45614C   C45631C
C45632C   B52004D   C55B07A   B55B09C   B86001W
CD7101F

e. C45531M..P (4 tests) and C45532M..P (4 tests) are inapplicable because the value of SYSTEM.MAX_MANTISSA is less than 48.

f. C47004A is expected to raise CONSTRAINT_ERROR whilst evaluating the comparison on line 51, but this compmiler evaluates the result without invoking the basic operation qualification (as allowed by 11.6 (7) LRM) which would raise CONSTRAINT_ERROR and so no exception is raised.

g. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

h. B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER or SHORT_INTEGER.

i. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

j. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, SHORT_FLOAT or LONG_FLOAT.

k. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

l. CD1009C, CD2A41A, CD2A41B, CD2A41E and CD2A42A..J (10 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for floating point types.

m. CD2A61I and CD2A61J are not applicable because this implementation imposes restrictions on 'SIZE length clauses for array types.

n. CD2A71A..D (4 tests), CD2A72A..D (4 tests), CD2A74A..D (4 tests) and CD2A75A..D (4 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for record types.

o. CD2A84B..I (8 tests), CD2A84K and CD2A84L are not applicable because this implementation imposes restrictions on 'SIZE length clauses for access types.

p. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

q. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

r. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

s. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

t. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

u. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

v. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

w. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

x. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

y. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

z. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

aa. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

ab. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

ac. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

ad. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

ae. CE2107C..D (2 tests) raise USE_ERROR when the function NAME is applied to temporary sequential files, which are not given names.

af. CE3102E is inapplicable because text file CREATE with IN_FILE mode is supported by this implementation.

ag. CE3102F is inapplicable because text file RESET is supported by this implementation.

ah. CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.

ai. CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.

aj. CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.

ak. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is not supported by this implementation.

al. CE3111B and CE3115A are inapplicable because they assume that a PUT operation writes data to an external file immediately. This implementation uses line buffers; only complete lines are written to an external file by a PUT_LINE operation. Thus attempts to GET data before a PUT_LINE operation in these tests raise END_ERROR.

am. CE3112B is inapplicable because, for this implementation, temporary text files are not given names.

an. CE3202A is inapplicable because the underlying operating system does not allow this implementation to support the NAME operation for STANDARD_INPUT and STANDARD_OUTPUT. Thus the calls of the NAME operation for the standard files in this test raise USE_ERROR.

ao. EE2401D contains instantiations of package DIRECT_IO with unconstrained array types. This implementation raises USE_ERROR upon creation of such a file.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 14 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A B24009A B29001A B38003A B38009A B38009B
B51001A B91001H BA1101E BC2001D BC2001E BC3204B
BC3205B BC3205D

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

No prevalidation results were submitted by the customer, because the results were nearly identical to those obtained for a prevalidation for the same customer.

3.7.2 Test Method

Testing of the SYSTEAM Ada Compiler SUN/SUNOS Version 1.81 using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host and Target computer:	Sun 3/60
Host and Target operating system:	SunOS, Version 4.0.3
Compiler:	SYSTEAM Ada Compiler SUN/SUNOS Version 1.81

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the Sun 3/60. Results were transferred to a VAX 8350, via DECNET, where they were printed and evaluated.

The compiler was tested using command scripts provided by SYSTEAM KG and reviewed by the validation team. Tests were compiled using the command

```
$3181/sas compile -v -l <file name>
```

and linked with the command

```
$3181/sas link -v <test name> -ld -o <test name>.OUT
```

Chapter B tests were compiled with the full listing option. A full description of compiler and linker options is given in Apendix E.

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at SYSTEAM KG, Karlsruhe and was completed on Thursday, 2nd November 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

SYSTEAM KG has submitted the following Declaration of Conformance concerning the SYSTEAM Ada Compiler SUN/SUNOS, Version 1.81

Declaration of Conformance

Customer:	SYSTEAM KG
Ada Validation Facility:	IABG m. b. H., Abt. SZT
ACVC Version:	1.10

Ada Implementation

Ada Compiler Name:	SYSTEAM Ada Compiler SUN/SUNOS
Version:	1.81
Host Computer System:	Sun 3/60 under SunOS, Version 4.0.3
Target Computer System:	Sun 3/60 under SunOS, Version 4.0.3

Customer's Declaration

I, the undersigned, representing SYSTEAM KG, declare that SYSTEAM KG has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration.

Signature

Karlsruhe, 3. November 1989
Date

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the SYSTEAM Ada Compiler SUN/SUNOS Version 1.81, as described in this Appendix, are provided by SYSTEAM KG. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD are also a part of this Appendix.

4 Predefined Language Environment

The predefined language environment comprises the package standard, the language-defined library units and the implementation-defined library units.

4.1 The Package STANDARD

The specification of the package standard is outlined here; it contains all predefined identifiers of the implementation.

The operations defined for the predefined types are not mentioned here, since they are implicitly declared according to the language rules. Anonymous types (such as *universal_integer*) are not mentioned either.

```
PACKAGE standard IS

   TYPE boolean IS (false, true);

   TYPE short_integer IS RANGE - 32_768 .. 32_767;

   TYPE integer IS RANGE - 2_147_483_648 .. 2_147_483_647;

   TYPE short_float IS DIGITS 6 RANGE
           - 16#0.FFFF_FF#E32 .. 16#0.FFFF_FF#E32;

   TYPE float IS DIGITS 15 RANGE
           - 16#0.FFFF_FFFF_FFFF_F8#E256 .. 16#0.FFFF_FFFF_FFFF_F8#E256;


   TYPE long_float IS DIGITS 18 RANGE
           - 16#0.FFFF_FFFF_FFFF_FFFF#E4096 ..
             16#0.FFFF_FFFF_FFFF_FFFF#E4096;

-- TYPE character IS ... as in [Ada,Appendix C]

-- FOR character USE ... as in [Ada,Appendix C]

-- PACKAGE ascii IS ... as in [Ada,Appendix C]
-- Predefined subtypes and string types ... as in [Ada.Appendix C]

   TYPE duration IS DELTA 2#1.0#E-14 RANGE
           - 131_072.0 .. 131_071.999_938_964_843_75;

-- The predefined exceptions are as in [Ada.Appendix C]

END standard;
```

4.2 Language-Defined Library Units

The following language-defined library units are included in the master library:

The package system
The package calendar
The generic procedure unchecked_deallocation
The generic function unchecked_conversion
The package io_exceptions
The generic package sequential_io
The generic package direct_io
The package text_io
The package low_level_io

4.3 Implementation-Defined Library Units

The master library also contains the implementation-defined library units collection_manager, timing, command_arguments and text_io_extension.

6 Appendix F

This chapter, together with Chapters 7 and 8, is the Appendix F required in [Ada], in which all implementation-dependent characteristics of an Ada implementation are described.

6.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

6.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here. All the other pragmas listed in Appendix B of [Ada] are implemented and have the effect described there.

CONTROLLED
has no effect.

INLINE
Inline expansion of subprograms is supported with following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

INTERFACE

is supported for C and assembler.

For each Ada subprogram for which

```
PRAGMA interface (C, <ada_name>);
```

is specified, a routine implementing the body of the subprogram <ada_name> must be provided, written in any language that obeys the C calling conventions (cf. [SunOS, C Programmer's Guide, Chapter D.3]), in particular:

- Saving registers
- Calling mechanism
- C stack frame format.

SunOS system calls or subroutines are allowed too.

The following parameter and result types are supported:

C Type	Ada Type
int	standard.integer
double	standard.float
pointer	system.address

The calling mechanism for all parameter types is call by value. The type **address** may serve to implement all kinds of call by references: The user may build all kinds of objects and pass their addresses to the C subprogram or SunOS system routine.

If

```
PRAGMA interface (assembler, <ada_name>);
```

is specified, a routine implementing the body of the subprogram <ada_name> and obeying the internal calling conventions of the SYSTEAM Ada Compiler must be provided.

The name of the routine which implements the subprogram <ada_name> should be specified using the pragma **external_name** (see §6.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved reference during linking.

These generated names are prefixed with an underline as well as external symbols of C (for example SunOS system calls); therefore the user should not define names beginning with an underline.

The following example shows the intended usage of the pragma `interface (C)` to call a SunOS system routine. The given procedure serves to open a file with a fixed name. It is called in the body of the main program.

```
WITH system;

PROCEDURE unix_call IS

   read_mode : CONSTANT integer := 8#0#;

   file_name : CONSTANT string := "/HO/TEST/F1" & ascii.nul;
   PRAGMA resident (file_name);

   ret_code  : integer;

   use_error : EXCEPTION;

   FUNCTION unix_open (path  : system.address;
                       oflag : integer) RETURN integer;
   PRAGMA interface (C, unix_open);
   PRAGMA external_name ("_open", unix_open);

BEGIN
   ret_code := unix_open (file_name'address, read_mode);
   IF ret_code = -1 THEN
      RAISE use_error;
   END IF;
END unix_call;
```

MEMORY_SIZE
has no effect.

OPTIMIZE
has no effect.

PACK
see §7.1.

PRIORITY
There are two implementation-defined aspects of this pragma: First, the range of the subtype priority, and second, the effect on scheduling (§5) of not giving this pragma for a task or main program. The range of subtype priority is 0 .. 15, as declared in the predefined library package system (see §6.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving pragma priority for it is the same as if the pragma priority 0 had been given (i.e. the task has the lowest priority). Moreover, in this implementation the package system must be named by a with clause of a compilation unit if the predefined pragma priority is used within that unit.

SHARED
is supported.

STORAGE_UNIT
has no effect.

SUPPRESS
has no effect, but see §6.1.2 for the implementation-defined pragma suppress_all.

SYSTEM_NAME
has no effect.

6.1.2 Implementation-Defined Pragmas

EXTERNAL_NAME (<string>, <ada_name>)

<ada_name> specifies the name of a subprogram or of an object declared in a library package, <string> must be a string literal. It defines the external name of the specified subprogram. The Compiler uses a symbol with this name in the `call` instruction for the subprogram. The subprogram declaration of <ada_name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to the subprogram which is declared last.

Upper and lower case letters are distinguished within <string>, i.e.<string> must be given exactly as it is to be used by external routines. The user should not define external names beginning with an underline because Compiler generated names as well as external symbols of C (for example SunOS system calls) are prefixed with an underline.

This pragma will be used in connection with the pragmas `interface (C)` or `interface (assembler)` (see §6.1.1).

RESIDENT (<ada_name>)

this pragma causes the value of the object <ada_name> to be held in storage (it may be held in a register too) and prevents assignments of a value to the object from being eliminated by the optimizer (see §3.2) of the SYSTEAM Ada Compiler. The following code sequence demonstrates the intended usage of the pragma:

```
...
x : integer;
a : SYSTEM.address;
  ...
BEGIN
  x := 5;
  a := x'ADDRESS;
  do_something (a);   -- let do_something be a non-local
                      -- procedure
                      -- a.ALL will be read in the body
                      -- of do_something
  x := 6;
  ...
```

If this code sequence is compiled by the SYSTEAM Ada Compiler without giving the option -O the statement `x := 5;` will be eliminated because from the point of view of the optimizer the value of x is not used before the next assignment to x. Therefore

```
PRAGMA resident (x);
```

should be inserted after the declaration of x.

This pragma can be applied to all those kinds of objects for which the address clause is supported (cf. §7.5).
It will often be used in connection with the pragma `interface (C, ... )` (see §6.1.1).

SQUEEZE
see §7.1.

SUPPRESS_ALL
causes all the run_time checks described in [Ada,§11.7] to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

6.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this chapter.

6.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in [Ada]. We note here only the implementation-dependent aspects.

ADDRESS
If this attribute is applied to an object for which storage is allocated, it yields the address of the first storage unit that is occupied by the object.
If it is applied to a subprogram or to a task, it yields the address of the entry point of the subprogram or task body.
If it is applied to a task entry for which an address clause is given, it yields the address given in the address clause.

For any other entity this attribute is not supported and will return the value system.address_zero.

IMAGE

The image of a character other than a graphic character (cf. [Ada, §3.5.5(11)]) is the string obtained by replacing each italic character in the indication of the character literal (given in [Ada, Annex C(13)]) by the corresponding upper-case character. For example, character'image(*nul*) = "NUL".

MACHINE_OVERFLOWS

Yields true for each fixed point type or subtype and for each floating point type or subtype.

MACHINE_ROUNDS

Yields true for each fixed point type or subtype and for each floating point type or subtype.

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows:
If a length specification (STORAGE_SIZE, see §7.2) has been given for that type (static collection), the attribute delivers that specified value.
In case of a dynamic collection, i.e. no length specification by STORAGE_SIZE given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed.
If the collection manager (cf. §4.3.1) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.

The value delivered by this attribute applied to a task type or task object is as follows:
If a length specification (STORAGE_SIZE, see §7.2) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned.

6.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

6.3 Specification of the Package SYSTEM

The package system required in [Ada,§13.7] is reprinted here with all implementation-dependent characteristics and extensions filled in.

```
PACKAGE system IS

   TYPE designated_by_address IS LIMITED PRIVATE;

   TYPE address IS ACCESS designated_by_address;
   FOR address'storage_size USE 0;

   address_zero : CONSTANT address := NULL;

   TYPE name IS (sun3_sunos);
   system_name  : CONSTANT name := sun3_sunos;

   storage_unit : CONSTANT := 8;
   memory_size  : CONSTANT := 2 ** 31;
   min_int      : CONSTANT := - 2 ** 31;
   max_int      : CONSTANT := 2 ** 31 - 1;
   max_digits   : CONSTANT := 18;
   max_mantissa : CONSTANT := 31;
   fine_delta   : CONSTANT := 2.0 ** (-31);
   tick         : CONSTANT := 1.0/50.0;

   SUBTYPE priority IS integer RANGE 0 .. 15;

   TYPE interrupt_number IS RANGE 1 .. 31;

   interrupt_vector : ARRAY (interrupt_number) OF address;
   -- The mapping of signal numbers to interrupt addresses is
   -- defined by this array.

   sighup   : CONSTANT := 1;
   sigint   : CONSTANT := 2;
   sigquit  : CONSTANT := 3;
   sigill   : CONSTANT := 4;
   sigtrap  : CONSTANT := 5;
   sigabrt  : CONSTANT := 6;
   sigemt   : CONSTANT := 7;
   sigfpe   : CONSTANT := 8;
   sigkill  : CONSTANT := 9;
```

```
sigbus    : CONSTANT := 10;
sigsegv   : CONSTANT := 11;
sigsys    : CONSTANT := 12;
sigpipe   : CONSTANT := 13;
sigalrm   : CONSTANT := 14;
sigterm   : CONSTANT := 15;
sigurg    : CONSTANT := 16;
sigstop   : CONSTANT := 17;
sigtstp   : CONSTANT := 18;
sigcont   : CONSTANT := 19;
sigchld   : CONSTANT := 20;
sigttin   : CONSTANT := 21;
sigttou   : CONSTANT := 22;
sigio     : CONSTANT := 23;
sigxcpu   : CONSTANT := 24;
sigxfsz   : CONSTANT := 25;
sigvtalrm: CONSTANT := 26;
sigprof   : CONSTANT := 27;
sigwinch  : CONSTANT := 28;
siglost   : CONSTANT := 29;
sigusr1   : CONSTANT := 30;
sigusr2   : CONSTANT := 31;


FUNCTION "+" (left : address; right : integer) RETURN address;

FUNCTION "+" (left : integer; right : address) RETURN address;

FUNCTION "-" (left : address; right : integer) RETURN address;

FUNCTION "-" (left : address; right : address) RETURN integer;

FUNCTION symbolic_address (symbol : string) RETURN address;

SUBTYPE external_address IS STRING;

-- External addresses use hexadecimal notation with characters
-- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:
--     "7FFFFFFF"
--     "80000000"
--     "8" represents the same address as "00000008"

FUNCTION convert_address (addr : external_address) RETURN address;

   -- CONSTRAINT_ERROR is raised if the external address ADDR
   -- is the empty string, contains characters other than
   -- '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address
   -- value cannot be represented with 32 bits.

FUNCTION convert_address (addr : address) RETURN external_address;

   -- The resulting external address consists of exactly 8
   -- characters '0'..'9', 'A'..'F'.


non_ada_error        : EXCEPTION;

-- non_ada_error is raised, if some event occurs which does not
-- correspond to any situation covered by Ada, e.g.:
--    illegal instruction encountered
--    error during address translation
--    illegal address

TYPE exception_id IS NEW integer;

no_exception_id      : CONSTANT exception_id := 0;

-- Coding of the predefined exceptions:

constraint_error_id  : CONSTANT exception_id := ... ;
numeric_error_id     : CONSTANT exception_id := ... ;
program_error_id     : CONSTANT exception_id := ... ;
storage_error_id     : CONSTANT exception_id := ... ;
tasking_error_id     : CONSTANT exception_id := ... ;

non_ada_error_id     : CONSTANT exception_id := ... ;

status_error_id      : CONSTANT exception_id := ... ;
mode_error_id        : CONSTANT exception_id := ... ;
name_error_id        : CONSTANT exception_id := ... ;
use_error_id         : CONSTANT exception_id := ... ;
device_error_id      : CONSTANT exception_id := ... ;
end_error_id         : CONSTANT exception_id := ... ;
data_error_id        : CONSTANT exception_id := ... ;
layout_error_id      : CONSTANT exception_id := ... ;

time_error_id        : CONSTANT exception_id := ... ;


TYPE exception_information IS
   RECORD
      excp_id          : exception_id;

         -- Identification of the exception. The codings of
```

```
            -- the predefined exceptions are given above.

         code_addr        : address;

            -- Code address where the exception occured. Depending
            -- on the kind of the exception it may be be address of
            -- the instruction which caused the exception, or it
            -- may be the address of the instruction which would
            -- have been executed if the exception had not occured.

          error_code       : integer;

      END RECORD;

   PROCEDURE get_exception_information
                (excp_info : OUT exception_information);

      -- The subprogram get_exception_information must only be called
      -- from within an exception handler BEFORE ANY OTHER EXCEPTION
      -- IS RAISED. It then returns the information record about the
      -- actually handled exception.
      -- Otherwise, its result is undefined.

   TYPE exit_code IS NEW integer;

   error         : CONSTANT exit_code :=  1;
   success       : CONSTANT exit_code :=  0;

   errno         : integer;
      -- error number of the last not successful system call.
      -- The value is that of the external variable errno
      -- of SunOS (cf. [SunOS,intro(2)]).

   PROCEDURE set_exit_code (val : exit_code);

      -- Specifies the exit code which is returned to the
      -- operating system if the Ada program terminates normally.
      -- The default exit code is 'success'. If the program is
      -- abandoned because of an exception, the exit code is
      -- 'error'.

PRIVATE

   -- private declarations

END system;
```

6.4 Restrictions on Representation Clauses

See Chapter 7 of this manual.

6.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §7.4 of this manual).

6.6 Expressions in Address Clauses

See §7.5 of this manual.

6.7 Restrictions on Unchecked Conversions

The implementation supports unchecked type conversions for all kind of source and target types with the restriction that the target type must not be an unconstrained array type. The result value of the unchecked conversion is unpredictable if

```
   target_type'SIZE > source_type'SIZE
```

6.8 Characteristics of the Input-Output Packages

The implementation-dependent characteristics of the input-output packages as defined in Chapter 14 of [Ada] are reported in Chapter 8 of this manual.

6.9 Requirements for a Main Program

A main program must be a parameterless library procedure. This procedure may be a generic instantiation; the generic procedure need not be a library unit.

6.10 Unchecked Storage Deallocation

The generic procedure unchecked_deallocation is provided; the effect of calling an instance of this procedure is as described in [Ada, §13.10.1].

The implementation also provides an implementation-defined package collection_manager, which has advantages over unchecked deallocation in some applications (cf. §4.3.1).

Unchecked deallocation and operations of the collection_manager can be combined as follows:

- collection_manager.reset can be applied to a collection on which unchecked deallocation has also been used. The effect is that storage of all objects of the collection is reclaimed.
- After the first unchecked_dealloation (release) on a collection, all following calls of release (unchecked deallocation) until the next reset have no effect, i.e. storage is not reclaimed.
- after a reset a collection can be managed by mark and release (resp. unchecked_deallocation) with the normal effect even if it was managed by unchecked_deallocation (resp. mark and release) before the reset.

6.11 Machine Code Insertions

A package machine_code is not provided and machine code insertions are not supported.

6.12 Numeric Error

The predefined exception numeric_error is never raised implicitly by any predefined operation; instead the predefined exception constraint_error is raised.

7 Appendix F: Representation Clauses

In this chapter we follow the section numbering of Chapter 13 of [Ada] and provide notes for the use of the features described in each section.

7.1 Pragmas

PACK

As stipulated in [Ada,§13.1], this pragma may be given for a record or array type. It causes the Compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type the pragma pack has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. All components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of `system.storage_unit`. Thus, the pragma pack does not effect packing down to the bit level (for this see pragma `squeeze`).

SQUEEZE

This is an implementation-defined pragma which takes the same argument as the predefined language pragma pack and is allowed at the same positions. It causes the Compiler to select a representation for the argument type that needs minimal storage space (packing down to the bit level). For components whose type is an array or record type the pragma `squeeze` has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. The components of a squeezed data structure will not in general start at storage unit boundaries.

7.2 Length Clauses

SIZE

for all integer, fixed point and enumeration types the value must be <= 32;
for short_float types the value must be = 32 (this is the amount of storage which is associated with these types anyway);
for float types the value must be = 64 (this is the amount of storage which is associated with these types anyway).
for long_float types the value must be = 96 (this is the amount of storage which is associated with these types anyway);
for access types the value must be = 32 (this is the amount of storage which is associated with these types anyway).
If any of the above restrictions are violated, the Compiler responds with a RESTRICTION error message in the Compiler listing.

STORAGE_SIZE

Collection size: If no length clause is given, the storage space needed to contain objects designated by values of the access type and by values of other types derived from it is extended dynamically at runtime as needed. If, on the other hand, a length clause is given, the number of storage units stipulated in the length clause is reserved, and no dynamic extension at runtime occurs.

Storage for tasks: The memory space reserved for a task is 10K bytes if no length clause is given (cf. Chapter 5). If the task is to be allotted either more or less space, a length clause must be given for its task type, and then all tasks of this type will be allotted the amount of space stipulated in the length clause (the activation of a small task requires about 1.4K bytes). Whether a length clause is given or not, the space allotted is not extended dynamically at runtime.

SMALL

there is no implementation-dependent restriction. Any specification for SMALL that is allowed by the LRM can be given. In particular those values for SMALL which are not a power of two are also supported .

7.3 Enumeration Representation Clauses

The integer codes specified for the enumération type have to lie inside the range of the largest integer type which is supported; this is the type integer defined in package standard.

7.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the Compiler responds with a RESTRICTION error message in the Compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1, 2 or 4 the starting address of an object will be a multiple of the specified alignment.

The number of bits specified by the range of a component clause must not be greater than the amount of storage occupied by this component. (Gaps between components can be forced by leaving some bits unused but not by specifying a bigger range than needed.) Violation of this restriction will produce a RESTRICTION error message.

There are implementation-dependent components of record types generated in the following cases :

- If the record type includes variant parts and the difference between the sizes of the maximum and the minimum variant is greater than 32 bytes, and, in addition, if it has either more than one discriminant or else the only discriminant may hold more than 256 different values, the generated component holds the size of the record object. (If the second condition is not fulfilled, the number of bits allocated for any object of the record type will be the value delivered by the size attribute applied to the record type.)
- If the record type includes array or record components whose sizes depend on discriminants, the generated components hold the offsets of these record components (relative to the corresponding generated component) in the record object.

But there are no implementation-generated names (cf. [Ada,§13.4(8)]) denoting these components. So the mapping of these components cannot be influenced by a representation clause.

7.5 Address Clauses

Address clauses are supported for objects declared by an object declaration and for single task entries. If an address clause is given for a subprogram, package or a task unit, the Compiler responds with a RESTRICTION error message in the Compiler listing.

If an address clause is given for an object, the storage occupied by the object starts at the given address. Address clauses for single entries are described in §7.5.1.

7.5.1 Interrupts

Under SunOS it is not possible to handle hardware interrupts directly within the Ada program; all hardware interrupts are handled by the operating system. In SunOS, asynchronous events are dealt with by signals [SunOS, sigvec (2)]. In the remainder of this section the terms *signal* and *interrupt* should be regarded as synonyms.

An address clause for an entry associates the entry with a signal. When a signal occurs, a signal catching handler, provided by the Ada runtime system, initiates the entry call.

By this mechanism, an interrupt acts as an entry call to that task; such an entry is called an *interrupt entry.* An interrupt causes the ACCEPT statement corresponding to the entry to be executed.

The interrupt is mapped to an *ordinary* entry call. The entry may also be called by an Ada entry call statement. However, it is assumed that when an interrupt occurs there is no entry call waiting in the entry queue. Otherwise, the program is erroneous and behaves in the following way:

- If an entry call stemming from an interrupt is already queued, this previous entry call is lost.
- The entry call stemming from the interrupt is inserted into the front of the entry queue, so that it is handled before any entry call stemming from an Ada entry call statement.

7.5.1.1 Association between Entry and Interrupt

The association between an entry and an interrupt is achieved via an interrupt number (type system.interrupt_number), the range of interrupt numbers being 1 .. 31 (this means that 31 single entries can act as interrupt entries). The meaning of the interrupt (signal) numbers is as defined in [SunOS, sigvec(2)]. A single parameterless entry of a task can be associated with an interrupt by an address clause (the Compiler does not check these conventions). Since an address value must be given in the address clause, the interrupt number has to be converted into type system.address. The array system.interrupt_vector is provided for this purpose; it is indexed by an interrupt number to get the corresponding address.

The following example associates the entry ir with signal SIGINT.

```
...
TASK handler IS

   ENTRY ir;

   FOR ir USE AT system.interrupt_vector (system.sigint);
END;
...
```

The task body contains ordinary accept statements for the entries.

7.5.1.2 Important Implementation Information

There are some important facts which the user of interrupt entries should know about the implementation. First of all, there are some signals which the user should not use within address clauses for entries. These signals are **sigfpe**, **sigsegv**, **sigbus** and **sigalrm**; they are used by the Ada Runtime System to implement exception handling and delay statements **sigalrm**. Programs containing address clauses for entries with these interrupt numbers are erroneous.

In the absence of address clauses for entries, the Ada Runtime System establishes signal catching handlers only for the signals mentioned above, so all other signals will lead to program abortion as specified in the SunOS documentation.

A signal catching handler for a specific signal is established when a task which has an interrupt entry for this signal is activated. The signal catching handler is deactivated und the previous handler is restored when the task has been completed. Several tasks with interrupt entries for the same signal may exist in parallel; in this case the signal catching handler is activated when the first of these tasks is created, and deactivated when the last of these tasks has been completed.

7.6 Change of Representation

The implementation places no additional restrictions on changes of representation.

8 Appendix F: Input-Output

In this chapter we follow the section numbering of Chapter 14 of [Ada] and provide notes for the use of the features described in each section.

8.1 External Files and File Objects

An external file is identified by a string that denotes a SunOS file name. It may consist of up to 1023 characters.

The form string specified for external files is described in §8.2.1.1.

8.2 Sequential and Direct Files

Sequential and direct files are ordinary files which are interpreted to be formatted with records of fixed or variable length. Each element of the file is stored in one record.

In case of a fixed record length each file element has the same size, which may be specified by a form parameter (see §8.2.1.1); if none is specified, it is determined to be $(\mathit{element\_type'SIZE} + \mathit{system.storage\_unit} - 1)/\mathit{system.storage\_unit}$.
In contrast, if a variable record length is chosen, the size of each file element may be different. Each file element is written with its actual length. When reading a file element its size is determined as follows:

- If an object of the `element_type` has a size component (see §7.4) the element size is determined by first reading the corresponding size component from the file.
- If `element_type` is constrained, the size is the minimal number of bytes needed to hold a constrained object of that type.
- In all other cases, the size of the current file element is determined by the size of the variable given for reading.

8.2.1 File Management

Since there is a lot to say about this section, we shall introduce subsection numbers which do not exist in [Ada].

8.2.1.1 The NAME and FORM Parameters

The name parameter must be a SunOS file name. The function name will return a path name string which is the complete file name of the file opened or created. Each component of the file name (separated by "/") is truncated to 255 characters. Upper and lower case letters within the file name string are distinguished.

The syntax of the form parameter string is defined by:

```
form_parameter ::= [ form_specification { , form_specification } ]

form_specification ::=  keyword [ => value ]

keyword ::=  identifier

value   ::=  identifier | numeric_literal
```

For identifier and numeric_literal see [Ada,Appendix E]. Only an integer literal is allowed as *numeric_literal* (see [Ada,§2.4]). In an identifier or numeric_literal, upper and lower case letters are not distinguished.

In the following, the form specifications which are allowed for all files are described.

```
MODE => numeric_literal
```

This value specifies the access permission of an external file; it only has an effect in a create operation and is ignored in an open. Access rights can be specified for the owner of the file, the members of a group, and for all other users. numeric_literal has to be a three digit octal number.

The access permission is then interpreted as follows:

8#400#	read access by owner
8#200#	write access by owner
8#100#	execute access by owner
8#040#	read access by group
..	write/execute access by group, analogously
8#004#	read access by all others
..	write/execute access by others, analogously

Each combination of the values specified above is possible. The default value is 8#666#.
The definitive access permission is then determined by the SunOS System. It will be the specified value for MODE, except that no access right prohibited by the process's file mode creation mask (which may be set by the SunOS umask command, cf. [SunOS, sh(1)/umask(2)]) is granted. In other words, the value of each "digit" in the process's file mode creation mask is subtracted from the corresponding "digit" of the specified mode. For example, a file mode creation mask of 8#022# removes group and others write permission (i.e. the default mode 8#666# would become mode 8#644#).

The following form specification is allowed for sequential and direct files:

```
  RECORD_SIZE => numeric_literal
```

This value specifies the size of one element on the file (record size) in bytes. This form specification is only allowed for files with fixed record format. If the value is specified for an existing file it must agree with the value of the external file.

By default, $(element\_type'SIZE + system.storage\_unit - 1)/system.storage\_unit$ will be chosen as record size, if the evaluation of this expression does not raise an exception. In this case, the attempt to create or open a file will raise `use_error`.

If a fixed record format is used, all objects written to a file which are shorter than the record size are filled up. The content of this extended record area is undefined. An attempt to write an element which is larger than the specified record size will result in the exception `use_error` being raised. This can only happen if the record size is specified explicitly.

8.2.1.2 Sequential Files

A sequential file is represented by an ordinary file that is interpreted to be formatted with either fixed-length or variable-length records (this may be specified by the form parameter).

If a fixed record format is used, all objects written to a file which are shorter than the maximum record size are filled up. The content of this extended record area is undefined.

```
RECORD_FORMAT => VARIABLE | FIXED
```

This form specification is used to specify the record format. If the format is specified for an existing file it must agree with the format of the external file.

The default is variable record size. This means that each file element is written with its actual length. A read operation transfers exactly one file element with its actual length.

Fixed record size means that every record is written with the size specified as record size.

```
APPEND => FALSE | TRUE
```

If the form specification APPEND => TRUE is given for an existing file in an open for an output file, then the file pointer will be set to the end of the file after opening, i.e. the existing file is extended and not rewritten. This form specification is only allowed for an output file; it only has an effect in an open operation and is ignored in a create. By default the value FALSE is chosen.

```
TRUNCATE => FALSE | TRUE
```

If the form specification TRUNCATE => TRUE is given for an existing file in an open for an output file, then the file length is truncated to 0, i.e. the previous contents of the file are deleted. Otherwise the file is rewritten, i.e. if the amount of data written is less than the file size, data previously written will remain at the end of the file. This form specification is only allowed for an output file; it only has an effect in an open operation and is ignored in a create. By default the value TRUE is chosen.

The default form string for a sequential file is :

```
"RECORD_FORMAT => VARIABLE, APPEND => FALSE,  " &
"TRUNCATE      => TRUE,     MODE   => 8#666#  "
```

8.2.1.3 Direct Files

The implementation dependent type count defined in the package specification of `direct_io` has an upper bound of :

```
COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)
```

A direct file is represented by an ordinary file that is interpreted to be formatted with records of fixed length. If not explicitly specified, the record size is equal to $(element\_type'SIZE + system.storage\_unit - 1)/system.storage\_unit$.

The default form string for a direct file is :

```
"RECORD_SIZE => ...., MODE => 8#666#"
```

8.3 Text Input-Output

Text files are sequential character files and are represented by streams. (A stream is a file with associated buffering, cf. [SunOS, stdio(3S)].)

Each line of a text file consists of a sequence of characters terminated by a line terminator, i.e. an ASCII.LF character.

A page terminator is represented by an ASCII.FF character and is always preceded by a line terminator.

A file terminator is not represented explicitly in the external file; the end of the file is taken as a file terminator. A page terminator is assumed to precede the end of the file if there is not explicitly one as the last character of the file.

Output to a file and to a terminal differ in the following way: If the output refers to a terminal it is unbuffered, which means that each write request in an Ada program will appear on the terminal immediately. Output to other files is buffered, i.e several characters are saved up and written as a block.

Terminal input is always processed in units of lines.

8.3.1 File Management

Beside the mode specification (cf. §8.2.1.1) the following form specification is allowed:

```
APPEND => FALSE | TRUE
```

If the form specification APPEND => TRUE is given for an existing file in an open for an output file, then the file pointer will be set to the end of the file after opening, i.e. the existing file is extended and not rewritten. This form specification is only allowed for an output file; it only has an effect in an open operation and is ignored in a create. By default the value FALSE is chosen.

The default form string for a text file is :

```
"APPEND => FALSE,  MODE => 8#666#"
```

8.3.2 Default Input and Output Files

The standard input (resp. output) file is associated with the standard SunOS files **stdin** resp. **stdout** (cf. [SunOS, stdio(3S)]).
Writing to the SunOS standard error file **stderr** may be done by using the package **text_io_extension** (cf. §4.3.4).

8.3.3 Implementation-Defined Types

The implementation-dependent types count and field defined in the package specification of text_io have the following upper bounds :

```
COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)

FIELD'LAST = 512
```

8.4 Exceptions in Input-Output

For each of name_error, use_error, device_error and data_error we list the conditions under which that exception can be raised. The conditions under which the other exceptions declared in the package io_exceptions can be raised are as described in [Ada,§14.4].

NAME_ERROR

- in an open operation, if the specified file does not exist;
- if the **name** parameter in a call of the **create** or open procedure is not a legal SunOS file name string; i.e, if a component of the path prefix is not a directory.

USE_ERROR

- whenever an error occurred during an operation of the underlying SunOS system. This may happen if an internal error was detected, an operation is not possible for reasons depending on the file or device characteristics, a capacity limit is exceeded or for similar reasons;
- if the function **name** is applied to a temporary file or to the standard input or output file;
- if an attempt is made to write or read to/from a file with fixed record format a record which is larger than the record size determined when the file was opened (cf. §8.2.1.1); in general it is only guaranteed that a file which is created by an Ada program may be reopened and read successfully by another program if the file types and the form strings are the same;
- in a **create** or **open** operation for a file with fixed record format (direct file or sequential file with **form** parameter RECORD_FORMAT => FIXED) if no record size is specified and the evaluation of the size of the element type will raise an exception.

(For example, if direct_io or sequential_io is instantiated with an unconstrained array type.)

- if a given form parameter string does not have the correct syntax or if a condition on an individual form specification described in §§8.2-3 is not fulfilled;

DEVICE_ERROR

is never raised. Instead of this exception the exception use_error is raised whenever an error occurred during an operation of the underlying SunOS system.

DATA_ERROR

the conditions under which data_error is raised by text_io are laid down in [Ada].

In general, the exception data_error is not usually raised by the procedure read of sequential_io and direct_io if the element read is not a legal value of the element type because there is no information about the file type or form strings specified when the file was created.
An illegal value may appear if the package sequential_io or direct_io was instantiated with a different element_type or if a different form parameter string was specified when creating the file. It may also appear if reading a file element is done with a constrained object and the constraint of the file element does not agree with the constraint of the object.
If the element on the file is not a legal value of the element type the effect of reading is undefined. An access to the object that holds the element after reading may cause a constrained_error, storage_error or non_ada_error.

8.5 Low Level Input-Output

We give here the specification of the package low_level_io:

```
PACKAGE low_level_io IS

   TYPE device_type IS (null_device);

   TYPE data_type IS
      RECORD
         NULL;
      END RECORD;

   PROCEDURE send_control    (device : device_type;
                              data   : IN OUT data_type);

   PROCEDURE receive_control (device : device_type;
                              data   : IN OUT data_type);

END low_level_io;
```

Note that the enumeration type device_type has only one enumeration value, null_device; thus the procedures send_control and receive_control can be called, but send_control will have no effect on any physical device and the value of the actual parameter data after a call of receive_control will have no physical significance.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	254 * 'A' & '1'
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	254 * 'A' & '2'
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	127 * 'A' & '3' & 127 * 'A'

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	127 * 'A' & '4' & 127 * 'A'
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	252 * '0' & "298"
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	250 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	`"` & 127 * `A` & `"`
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	`"` & 127 * `A` & `1` & `"`
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	235 * ` `
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2_147_483_648
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	SUN3_SUNOS
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	512
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_FLOAT_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	0.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	200_000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	15
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	nodir/file1
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	wrondir/file2

Name and Meaning	Value
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-0.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-200_000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	18
$MAX_IN_LEN Maximum input line length permitted by the implementation.	255
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:" & 250 * `0` & "11:"
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:" & 248 * `0` & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	`"` & 253 * `A` & `"`
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	SUN3_SUNOS
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

Name and Meaning	Value
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2_147_483_648
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	SUN3_SUNOS
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	1.0/50.0

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

b. A39005G This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E This test contains an unitended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING-_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests] These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests] These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

i. CD2B15C & CD7205C These tests expect that a 'STORAGE_SIZE length clause provides precise con- trol over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C & D, ED7006C & D [5 tests] These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B, & CD7204B These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

q. CE3111C This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

s. CE3411B This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER AND LINKER OPTIONS

This appendix contains information concerning the compilation and linkage commands used within the command scripts for this validation.

3 Compiling, Linking and Executing a Program

3.1 Overview

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units can be placed on different source files or they can all be on the same file. One unit, a parameterless procedure, acts as main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be executed.

§3.2 and §3.4 describe in detail how to call the Compiler and the Linker. Further on in §3.3 the Completer, which is called to generate code for instances of generic units and to complete packages without bodies, is described.
§3.5 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.
The information the Compiler produces and outputs in the Compiler listing is explained in §3.6.
Finally, the protocol of a sample session is given in §3.7.

3.2 Starting the Compiler

To start the SYSTEAM Ada Compiler, call

```
ac [<options>] <source> ...
```

Options:

`-lib <directory>`	program library
`-l`	create listing(s)
`-L <directory>`	create listing(s) in <directory>
`-L <file>`	create a listing on <file>
`-s`	produce symbolic code listing(s)
`-S`	suppress all runtime checks
`-O`	switch off the optimizer
`-I`	suppress inline expansion
`-c`	copy the source files into the program library
`-v`	print additional Compiler messages

Analogously to all other commands of the SYSTEAM Ada System, the Compiler can also be started by typing

```
sas compile  [<options>]  <source> ...
```

The input files for the Compiler are specified by <source> ... . The maximum length of lines in the sources is 255; longer lines are cut and an error is reported.

Each source file may contain a sequence of compilation units, cf. §10.1 of [Ada]. All compilation units in a source file are compiled individually. When a compilation unit is compiled successfully, the program library is updated and the Compiler continues with the compilation of the next unit on the source file. If the compilation unit contained errors, error messages are send to standard error output unless the **-L** or **-l** option is given (see below); in this case they are reported in the listing file (see §3.6). No update operation is performed on the program library and all subsequent compilation units in the compilation are only analyzed without generating code.

The compiler accepts the following options:

-lib <directory> specifies the name of the program library; the name of the default program library is `adalib`. The library must exist (see §2.2 for information on program library management).

The options -l and **-L** produce listing(s) containing all source lines and messages generated by the Compiler. (See §3.6 for more information about the listing.)
If -l is given, a separate listing file is generated under the current working directory for each source using the default listing file name. It is determined by appending `.l` to the source file name after removing all the characters from the last "." to the end of the source file name.
-L <file> resp. <directory> specifies the listing file(s) to be used. If -L <directory> is specified, a separate listing file is generated under that directory for each source using the default listing file name. Option **-L** <file> will produce a single listing file <file> containing the concatenation of all listings.
If neither -l nor **-L** is specified and a compilation unit contained errors, the error messages of the Compiler are listed on standard error output.

A symbolic code listing will be produced for each source and appended to the listing if the option -s is specified when calling the Compiler. This option is ignored if neither -l nor **-L** is given.

The option **-S** has the same effect as the corresponding pragma `SUPPRESS_ALL` would have at the beginning of the source (see [Ada,Appendix B] and §6.1.2 of this manual).

Inline expansion of any subprograms which are specified by a pragma `inline` (cf. §6.1.1) in the Ada source can be suppressed by giving the option **-I**.

No optimizations like constant folding, dead code elimination or structural simplifications are done if the **-O** option is specified.

The -c option causes the Compiler to copy the source files <source> ... into the program library. The Debugger of the SYSTEAM Ada System (cf. [ST16/87]) can then work on these copies (cf. §2.2.7) instead of on the original files.

If the -v option is specified the following information is printed on standard error output:

- a start message of the Compiler
- the program library used by the compilation
- the name of the source file currently being compiled
- for each compilation unit:
  the kind and the name of the unit
  the number of error messages and warnings issued by the Compiler (see §3.6)
  the CPU time used by the Compiler

As described above, the error messages of the Compiler are also reported on standard error output if no listing file is created, independently of the -v option.

The Compiler returns the value 1 on termination if one of the compilation units contained errors; otherwise the value 0.

3.3 The Completer

Since the Compiler does not generate code for instances of generic bodies, the Completer must be used to complete such units before a program using such instances can be executed. The Completer must also be used to complete packages in the program which do not require a body. This is done implicitly when the Linker is called.

It is also possible to call the Completer explicitly by

```
sas complete [<options>] <ada_name>
```

Options:

`-lib <directory>`	program library
`-l`	create a listing file
`-L <directory>`	create a listing file in directory <directory>
`-L <file>`	create a listing file <file>
`-s`	produce a symbolic code listing
`-S`	suppress all runtime checks
`-O`	switch off the optimizer
`-I`	suppress inline expansion
`-v`	print additional Completer messages

<ada_name> must be the name of a library unit. All library units that are needed by that unit (cf. [Ada,§10.5]) are completed, if possible, and so are their subunits, the subunits of those subunits and so on. Options apply to all units that are completed. The meaning of the options corresponds to that of the COMPILE command (cf. §3.2).

If the completer detected some error, error messages are generated and send to standard error output unless the **-L** or **-l** option is given. In this case the listing file will contain the error messages (cf. §3.6). If option -l is specified, the listing file is created in the current working directory with name `complete.l`. **-L** <file> resp. <directory> specifies the listing file to be used. If a directory name is specified, the listing file will be `<directory>/complete.l`; if a file name is given, the listing will be put onto that file.
If the specified unit is already completed, no listing file will be generated even if the **-L** or -l option is specified.

If the **-v** option is specified the following information is printed on standard error output:

- a start message of the Completer

- the program library used
- the name of the unit currently being completed
- the number of error messages and warnings issued by the Completer
- the CPU time used by the Completer

The Completer returns the value 1 on termination if it detected any errors, otherwise the value 0.

3.4 The Linker

An Ada program is a collection of units used by a main program which controls the execution. The main program must be a parameterless library procedure; any parameterless library procedure within a program library can be used as a main program.

The SYSTEAM Ada System Linker combines all program units used by the main program to an object file, using the SunOS link editor **ld(1)**.

To link a program, call the command

```
sas link [<options>] <ada_name> [-ld <ld_options>]
```

Options:

`-lib <directory>`	program library
`-C`	suppress completion of the program
`-l`	create a listing file
`-L <directory>`	create a listing file in directory <directory>
`-L <file>`	create a listing file <file>
`-s`	produce a symbolic code listing
`-S`	suppress all runtime checks
`-O`	switch off the optimizer
`-I`	suppress inline expansion
`-d`	suppress generation of debugger information
`-t`	suppress selective linking
`-v`	print start messages
`-r`	suppress generation of an executable object

ld_options: Options recognized by the SunOS **ld(1)** command.

<ada_name> is the name of the library procedure which acts as the main program.

By default the file a.out contains the executable code after linking.

The Linker accepts the following options:

-lib <directory> specifies the name of the program library which contains the main program; adalib is assumed by default.

The -C option suppresses completion of the main program before it is linked. If -C is not set the Completer is called with the options -lib, -l, -L, -s, -S, -O and -I (if specified) (cf. §3.3). The options -S and -O also affect the Pre-Linker (see below) because it generates code.

The option **-r** prevents generation of an executable object file. In this case, the generated object file contains the code of all compilation units written in Ada and of those object modules of the predefined language environment and of the Ada runtime system which are used by the main program; references into the Standard C library remain unresolved. The generated object file is suitable for further ld processing. The name of its entry point is _main.

If the **-d** option is given no information for the SYSTEAM Debugger is generated.

The option **-t** causes the code of *all compilation units mentioned in a context* clause (in a transitive manner) to be linked together. If it is not specified, selective linking is performed, i.e. the object code of any subprogram body will be included in the executable program only if this subprogram may be called during program execution.

If the **-v** option is specified, start messages of the Completer, Pre-Linker and Linker (see below) are printed on standard error output.

-ld <ld_options> can be used to specify further options for the SunOS System Linker. Each option accepted by the SunOS System Linker is allowed. In particular, the option -o <file> may be used to specify the name of the object file. Moreover, <ld_options> may include names of object files or archive libraries which contain object modules of those program units which are not written in Ada (e.g. object modules of subprograms written in C or in assembly language). For those program units the pragmas

```
   PRAGMA interface (C, ... )        -- (cf. §6.1.1)
 (or interface (assembler, ... ) )
and

   PRAGMA external_name ( ... )      -- (cf. §6.1.1)
 must be given in the Ada source.
```

<ld_options> will be passed to the SunOS System Linker without examining its correctness.

The following steps are performed during linking: First the Completer is called, unless suppressed by the -C option, to complete the bodies of all instances which are used by the main program and all packages which are used by the main program and do not require a body. Then the Pre-Linker is executed; it determines the compilation units that have to be linked together and a valid elaboration order and generates a code sequence to perform the elaboration. Subsequently the Ada Linker produces one object module that contains the code of all compilation units written in Ada. The format of the object module is the `a.out` format. Finally, the SunOS System Linker ld(1) is invoked to combine the object module generated by the Ada Linker, the object modules of the predefined language environment and of the Ada runtime system which are used by the main program (these object modules are contained in an archive library which is searched by ld), and also object modules specified by <ld_options>, to one (executable) object module.

The call of the SunOS System Linker generated is

```
   ld [-N -Bstatic] -r obj rtslib ld_options
```

if the option `-r` is given, and

```
   ld [-N -Bstatic] -dc -dp -X -e start /lib/crt0.o \
                 obj rtslib ld_options -lc
```

otherwise. Here, *obj* denotes the file containing the object module which is produced by the Ada Linker and *rtslib* the above-mentioned archive library containing the Ada runtime system. (This may be `librtsdbg.a` resp. `librts.a` if the SYSTEAM Ada Linker is called with option -d. In this case the -N and -Bstatic options are missing.)

If you invoke `ld` by yourself to link the executable object rather than having the Ada Linker doing it automatically, then you must explicitly specify a startup module (see below) and any libraries you want linked into the Ada program. Furthermore, the `ld` option -N should be specified to allow the resulting object file to be debugged by the SYSTEAM Ada System Debugger. (Note that debugging is only possible if the option -d was not passed to the Ada Linker.)

The startup module must satisfy the following requirements:

- A global variable called `_environ` is defined containing a pointer to the current environment (cf. [SunOS,environ(5)]).
- The Ada main program is called using the entry point `_main`.
- `argc` and `argv` are passed as arguments to `_main`.

Note that instructions, following the call of `_main`, will never be executed.
By default, the Standard C startup routine `/lib/crt0.o` is used.

The Linker of the SYSTEAM Ada System returns the value 1 if one of the above steps fail (e.g. if some compilation unit cannot be found in the program library or if no valid elaboration order can be determined because of incorrect usage of the pragma `elaborate`).

3.5 Executing a Program

After linking, the Ada program can be executed by typing the name of the generated object module (which is a.out by default).

If an Ada program is abandoned due to an unhandled exception, a message is displayed (see below). A core image file **adacore** is produced if the option -d was not specified during linkage of the program, cf. §3.4. The contents of this file can be interpreted using the SYSTEAM Ada Debugger, cf. [ST16/87].

The message displayed if an Ada program is abandoned due to an unhandled exception has the following form:

```
(1)  *** Ada program abandoned due to unhandled exception!
(2)      exception   : ...
(3)      raised at   : ...
(4)      error code  : ...
```

In line (2) the exception identification is displayed. For the predefined and I/O exceptions, the Ada names are printed. For all user-defined exceptions, a hexadecimal value uuuuxxxx is shown: uuuu indicates the library key of the compilation unit in which the exception is declared, xxxx is the compilation unit relative number of the exception. Non_ada_error, defined in package **system**, stands for any other exception.

In line (3) a code address or the value 0 is shown. If an address is given this can be the address of the instruction that caused the exception or of the following instruction. A value of 0 indicates that an unsuccessful system call led to the exception.

If an unsuccessful system call during an I/O operation caused the Ada program to abandon - in this case the exception identification in line (2) denotes an I/O exception and the value displayed in line (3) is 0 - the error number **errno** (cf. [SunOS, intro(2)]) returned by this system call is printed in line (4). In any other case the number of the signal that raised the exception is displayed.

Special note: No shell variables with names ADA_DGB_HT_PIPF and ADA_DBG_TH_PIPE should be defined by the user when executing an Ada program which was linked without specifying the option -d. These shell variables are used in a Debugger session to establish the communication between the Ada program and the SYSTEAM Ada Debugger. If they are defined when an Ada program is running the program will try to communicate with a Debugger process even if no Debugger is started.

3.6 The Compiler Listing

By default, messages of the Compiler are listed on standard error output. A complete listing can be obtained on a file by using the Compiler option -l or **-L** (cf. §3.2). This file will contain the whole source together with the messages of the Compiler.

The listing for a compilation unit starts with the kind and the name of the unit and the library key of the current unit.

Example:

```
=  PROCEDURE  MAIN.   Library Index   76
```

The format effectors ASCII.HT, ASCII.VT, ASCII.CR, ASCII.LF and ASCII.FF are represented by a '~' character in the listing. In any case, those source lines which are included in the listing are numbered to make locating them in the source file easy.

Errors are classified into SYMBOL ERROR, SYNTAX ERROR, SEMANTIC ERROR, RESTRICTION, COMPILER ERROR, WARNING and INFORMATION:

SYMBOL ERROR
: pinpoints an inappropriate lexical element. "Inappropriate" can mean "inappropriate in the given context". For example, '2' is a lexical element of Ada, but it is not appropriate in the literal 2#012#.

SYNTAX ERROR
: indicates a violation of the Ada syntax rules as given in [Ada,Appendix E].

SEMANTIC ERROR
: indicates a violation of Ada language rules other than the syntax rules.

RESTRICTION
: indicates a restriction of this implementation. Examples are representation clauses which are provided by the language but are not supported in this implementation; or situations in which the internal storage capacity of the Compiler for some sort of entity is exceeded.

COMPILER ERROR
: We hope you will never see a message of this sort.

WARNING
: messages tell the user facts which are likely to cause errors (for example, the raising of exceptions) at runtime.

INFORMATION
: messages tell the user facts which may be useful to know but probably do not endanger the correct running of the program. Examples are that a library unit named in a context clause is not used in the current compilation unit, or that another unit (which names the current compilation unit in a context clause) is made obsolete by the current compilation.

Warnings and information messages have no influence on the success of a compilation. If there are any other diagnostic messages, the compilation was unsuccessful.

All error messages are self-explanatory. If a source line contains errors, the error messages for that source line are printed immediately below it. The exact position in the source to which an error message refers is marked by a number. This number is also used to relate different error messages given for one line to their respective source positions.

In order to enable semantic analysis to be carried out even if a program is syntactically incorrect, the Compiler corrects syntax errors automatically by inserting or deleting symbols. The source positions of insertions/deletions are marked with a vertical bar and a number. The number has the same meaning as above. If a larger region of the source text is affected by a syntax correction, this region is located for the user by repeating the number and the vertical bar at the end as well, with dots in between these bracketing markings.

A complete Compiler listing follows which shows the most common kinds of error messages, the technique for marking affected regions and the numbering scheme for relating error messages to source positions. It is slightly modified so that it fits into the page width of this document:

```
**********************************************************************
**                                                                  **
** SYSTEAM ADA - COMPILER         SUN3/SUNOS 1.81                   **
**                                                                  **
** 89-10-11 / 09:06:02                                              **
**                                                                  **
**********************************************************************

======================================================================
=                                                                    =
=                                         Started at  : 09:06:02     =
=                                                                    =
=  PROCEDURE  LISTING_EXAMPLE                                        =
=                                                                    =
     1        PROCEDURE listing_example IS
     2        abc : procedure integer RANGE 0 .. 9 := 10E-1;
                    |1......1|
                                                         1
>>>>>  SYNTAX ERROR
         Symbol(s)  deleted (1)
>>>>>  SYMBOL ERROR (1)    An exponent for an integer literal must not
                           have a minus sign
     3        def integer RANGE 0 .. 9;
                  |1
>>>>>  SYNTAX ERROR
         Symbol(s) inserted (1):  :
     4        bool : boolean;
     5        BEGIN
     6        bool := (abc AND (def * 1)) OR adf;
                         1       2           3
>>>>>  SEMANTIC ERROR (1)  Actual parameter for LEFT is not of
                           appropriate type
>>>>>  SEMANTIC ERROR (2)  Actual parameter for RIGHT is not of
                           appropriate type
>>>>>  SEMANTIC ERROR (3)  Identifier ADF not declared
     7        END;
=                                                                    =
=  PROCEDURE  LISTING_EXAMPLE                                        =
=                                                                    =
=  **** Number of Errors             :     6                         =
=  **** Number of Warnings           :     0                         =
=                                                                    =
=  **** Number of Source  Lines      :      7                        =
=  **** Number of Comment Lines      :      0                        =
=  **** Number of Lexical Elements   :    42                         =
=  **** Code Size in Bytes             :     0                             =
=  **** Number of Diana Nodes created :    51                             =
=  **** Symbol  Error  in Line         :      2.                          =
=  **** Syntax  Error  in Line         :      2,     3.                   =
=  **** Semantic Error in Line         :      6.                          =
=  **** CPU Time used                  :     1.3  Seconds                 =
=                                        Finished at : 09:06:04           =
=                                                                         =
=========================================================================
*************************************************************************
**                                                                     **
** End of  Ada  Compilation            CPU Time used :  1.3  Seconds  **
**                                                                     **
*************************************************************************
```

3.7 Sample Session: Compile, Link and Execute

This chapter shows the log of a sample session. The lines starting with "$" are SunOS commands, all other lines are output. It is assumed that the current working directory is /disk0/ada/test.

(For example2.a it is further assumed that a routine with the name ASSEMBLER_EXAMPLE, which outputs the text "Assembler routine is called", has been written in assembly language and that the file /disk0/ada/test/a.o contains its object code.)

```
$ sas create -v
SYSTEAM ADA  -  LIBRARY-MANAGER   SUN-3/SUNOS  1.81


$ ac  -l  -v example1.a
SYSTEAM ADA  -  COMPILER          SUN-3/SUNOS  1.81
Library:     /disk0/ada/test/adalib
Compiling:  /disk0/ada/test/example1.a
PROCEDURE  LISTING_EXAMPLE,   Library Index   45
  **** Number of Errors              :     6
  **** Number of Warnings            :     0
CPU Time used :     1.2  Seconds
```

```
$ cat example2.a

WITH text_io;
USE  text_io;

PROCEDURE execution_example IS
   PROCEDURE assembler_routine;
   PRAGMA interface (assembler, assembler_routine);
   PRAGMA external_name ("ASSEMBLER_EXAMPLE",
                          assembler_routine);
BEGIN
   put_line ("Main program starts");
   assembler_routine;
   put_line ("Main program stops");
END execution_example;
```

```
$ ac -v example2.a
SYSTEAM ADA   -  COMPILER          SUN-3/SUNOS  1.81
Library:    /disk0/ada/test/adalib
Compiling:  /disk0/ada/test/example2.a
PROCEDURE  EXECUTION_EXAMPLE,   Library Index   46
*** No Errors found ***
CPU Time used :    2.1  Seconds


$ sas link execution_example -ld a.o  -o example.out

$ example.out

Main program starts
Assembler routine is called
Main program stops


$ sas delete
** Information: Program library /disk0/ada/test/adalib deleted
```